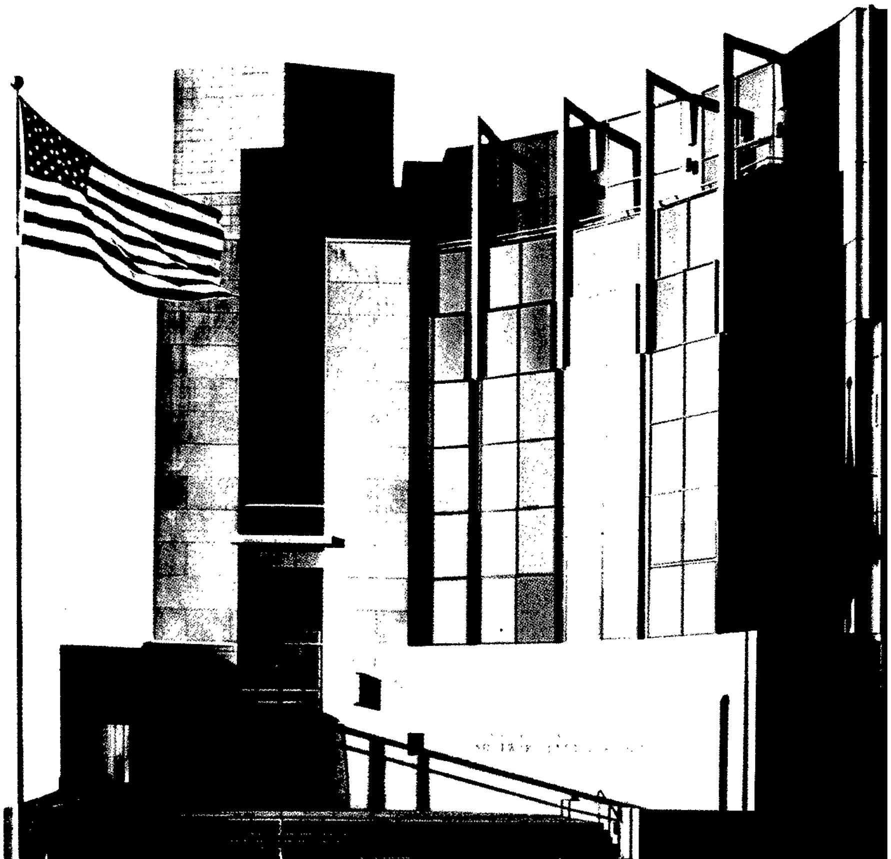
Carnegie Mellon
Software Engineering Institute

A Study of Practice Issues in Model-Based Verification Using the Symbolic Model Verifier (SMV)

19990114 019

Grama R. Srinivasan
David P. Gluch

November 1998



TECHNICAL REPORT
CMU/SEI-98-TR-013
ESC-TR-98-013

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

A Study of Practice Issues in Model-Based Verification Using the Symbolic Model Verifier (SMV)

CMU/SEI-98-TR-013
ESC-TR-98-013

Grama R. Srinivasan
David P. Gluch

November 1998

Dependable Systems Upgrade Initiative

Unlimited distribution subject to the copyright.

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Mario Moya

Mario Moya, Maj, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1998 by Carnegie Mellon University.

Requests for permission to reproduce this document or to prepare derivative works of this document should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

This document is available through Asset Source for Software Engineering Technology (ASSET): 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 or toll-free in the U.S. 1-800-547-8306 / FAX: (304) 284-9001 World Wide Web: http://www.asset.com / e-mail: sei@asset.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218 / Phone: (703) 767-8274 or toll-free in the U.S.: 1-800 225-3842.

DTIC QUALITY INSPECTED 3

Table of Contents

Abstract vii

Acknowledgements ix

1 Introduction xi

2 Background 1
- 2.1 Model-Based Verification 1
- 2.2 The Simplex Coordinated Demonstration System 2

3 Modeling with Statecharts 5
- 3.1 Statechart Notation 5
- 3.2 Creating the Statechart Model 6

4 Modeling Checking with the Symbolic Model Verifier (SMV) 9
- 4.1 Model Checking Steps 9
- 4.2 SMV Modeling Notation: 10
- 4.3 Creating the SMV Model 11
  - 4.3.1 Learning SMV 11
  - 4.3.2 Variables 12
  - 4.3.3 Model Refinement 12
  - 4.3.4 Iterations of the Model 13
  - 4.3.5 Syntax Checking 14
- 4.4 Checking the Claims 14
  - 4.4.1 Claim 1 15
  - 4.4.2 Claim 2 15
  - 4.4.3 Claim 3: 16

5 Process Metrics and Observations 19
- 5.1 Learning the System 19
- 5.2 Learning SMV Modeling Language and Tool 20
- 5.3 Modeling the System 20
- 5.4 Making Changes to the Models 21
- 5.5 Generating and Checking Claims 22

6 Summary **23**
6.1 Observations on the Modeling Effort 23
6.2 Observations on the Practice 24
6.3 Observations on Applicability 25
6.4 Future Work 25

7 References **27**

Appendix A: Glossary **31**

Appendix B: State Model of the System **33**

Appendix C: An Excerpt of the Specification **37**

List of Figures

Figure 1: The Coordinated Demonstration System 2
Figure 2: Concurrent Models of the Outer Loop 3
Figure 3: State Notation Used in the Case Study 5

Figure 4: Refined Experimental Coordinator Statechart 7
Figure 5: The SMV Model Checking Process 9
Figure 6: Ready-Busy Example 11

Figure 7: SMV Model of the Ready-Busy Example 12
Figure 8: The SMV Representation of the Experimental Coordinator 13
Figure 9: Safety Coordinator States 14

Figure B1: Overall Configuration of the System 33
Figure B2: States of the Overall System 33
Figure B3: States of the Decision Unit 34

Figure B4: States of the Safety Controller 34
Figure B5: States of the Experimental Coordinator 35
Figure B6: States of the Baseline Coordinator 35

List of Tables

Table 1: SMV Trace for Claim 2 16
Table 2: Percent Time in Each Major Activity 19

Abstract

This report presents the results of a study on the practice issues involved in using the Symbolic Model Verifier (SMV) for model checking software systems. The case study is of a Simplex implementation—the Simplex coordinated demonstration system for reliable system upgrade. The investigation consisted of generating a system model (using both statechart and SMV notations), specifying claims (expected properties) of the system as temporal logic formulae, and checking those formulae with respect to the SMV model. The various steps involved in the modeling process are described. Examples of the claims, their results, and a description of how the SMV tool analyzed them are detailed. Key engineering decisions made during the modeling process and a work breakdown of the effort are also presented.

Acknowledgements

The authors would like to express their thanks to Dr. Danbing Seto, who developed the design specifications used in this study, for his insightful observations and invaluable discussions throughout this effort. We also would like to thank Jared Brockway, Peter Feiler, Francisco Valeriano, and Charles Weinstock for their suggestions and constructive reviews of this work. Douglas Mosurak helped us to prepare the final manuscript and his help is gratefully acknowledged.

The first author would especially like to express his gratitude to Dr. James E. Tomayko, Director of the Master of Software Engineering program, for his encouragement and support. He would also like to express his appreciation to Sergey Berezin of the SMV design group for helping him to understand several subtleties of SMV and to Mrs. Phyllis A. Lewis, MSE administrator, for the guidance and moral support she provided.

This work was undertaken as an independent study project by the first author, Grama R. Srinivasan. The independent study was completed in partial fulfillment of the requirements for a Master of Software Engineering degree at Carnegie Mellon University.

1 Introduction

Modeling, particularly mathematical modeling, is an integral part of other engineering disciplines. There are numerous examples in the literature demonstrating the effectiveness of modeling in engineering endeavors [Jackson 98, Parnas 98]. For example, in order to build a bridge, an engineer creates drawings, formal engineering documents, mathematical models, and physical models and interacts with other engineers using these artifacts. The approval for and construction of the bridge is based in large part on the engineering models created in the early part of the design effort. After the bridge is in use, even for several decades, if a change is to be made in the design of the bridge, the engineers consult the original documentation and models and, before making changes, use these artifacts to justify these changes. Similar processes are followed in the aircraft industry, where structural and aerodynamic models of the airframe are created and analyzed before any change is made to the physical structure.

There are varying views about how to incorporate formalism and models in software engineering [Clarke 96b, Jackson 96, Jackson 98, Parnas 98] but often today, for software systems, the code is directly modified without consulting design or related engineering documentation. Though it is prone to error, this approach can work for a short time for uncomplicated systems, but the cumulative adverse effects can ultimately be very costly. For complex systems though, this approach is problematic. Realistic software systems are extremely complex and the source code should not be the only engineering model of the system that is used to engineer an upgrade. Models, engineering artifacts that faithfully reflect the system but that step back from implementation, can aid in reasoning about the properties of the system, provide rationale, and support subsequent system upgrades.

By presenting the results of a case study, this report explores practice issues involved in the use of the Symbolic Model Verifier (SMV) for model checking of a system. The Symbolic Model Verifier (SMV) is a model checking tool [Clarke 95, McMillan 92] that accepts a finite state representation of a system and a set of properties or claims made about that system in Computational Tree Logic (CTL). The SMV tool checks whether these properties hold for the model. In most cases, it also provides counterexamples for the properties that do not hold. The case study is of the Simplex coordinated demonstration system, an implementation of the Simplex architecture [Sha 96, Simplex 98, STR 98].

As an emerging discipline, software engineering is evolving into one that increasingly relies on models as a basis for engineering practice. This work is intended to contribute to the evolution of software engineering as a discipline and provide additional data on the use of modeling within software engineering practice.

Section 1 provides a background for the case study by presenting model-based verification, model-checking techniques, Simplex, and the Simplex coordinated demonstration system. Section 2 describes the generation of statechart models for the case study. An overview of SMV, its use in creating system models, and the issues involved in the model checking with SMV are presented in section 3. Section 4 describes process issues, including a summary of the various activities conducted in the case study. A summary of the observations, results, and future work are presented in Section 5. The appendices provide a glossary of terms and acronyms, the details of the model developed in this study, and an excerpt from the specification used in the study.

2 Background

This work was undertaken as of an independent study by the first author. The principal objectives of the study were to capture process and engineering data and to investigate feasibility issues in the application of model-based approaches for verifying (finding errors) in software systems. The technical focus of the modeling effort was to explore the fault tolerance responses of the Simplex coordinated demonstration system.

2.1 Model-Based Verification

Model-based verification [Gluch 98] is a verification practice for upgrades that relies upon the creation and analysis of essential models of a system. Essential models are abstracted representations of requirements, design, or code that capture the critical aspects of a system. Model-based verification techniques encompass a variety of modeling approaches including many that have been identified as lightweight formal methods [Jackson 96].

Modeling a system is a process of forming a representation of the system that shares important properties with that system [Jackson 95]. The model is generally an abstraction of the system and is often used to help reason about some interesting or important properties of the system. It also serves as a common framework for understanding and communication among system developers and users.

One embodiment of the model-based verification approach is termed "model checking" and has been successfully applied in hardware and protocol analyses. In model checking, an abstract formal model is created and checked, using an automated tool, against desired (required) properties. The success stories for model checking in the hardware and protocol analysis fields can be found in the literature [Clarke 95, Clarke 96a, Clarke 96b, Fujita 96, Raimi 97].

Model checking techniques rely principally on state machine models. State machine models of practical (realistic size) software systems are complicated by the problem of state-explosion [McMillan 92]. This size explosion results from the fact that the total number of combinations of possible states that a system can occupy during its use is extremely large—hundreds of thousands of states or more. Hence, trying to represent all valid states individually and to identify the transitions among them is impractical. Modeling and analysis techniques have been and are being developed to alleviate this problem [Clarke 86, Clarke 96a, McMillan 92]. The Symbolic Model Verifier (SMV) tool used in this study employs symbolic rather than explicit state representation to address the state explosion problem.

2.2 The Simplex Coordinated Demonstration System

The case study for this investigation is the Simplex architecture coordinated demonstration system. The physical configuration of the coordinated demonstration system is shown in Figure 1. It consists of two carts, each of which supports an inverted pendulum. There is a rod extending between the top of each pendulum. Each cart moves back and forth along its own track. The control goal of the system is to balance each pendulum and the rod between them as the carts move along.

The control functions for the system are partitioned into inner loop controllers and outer loop coordinator modules. The inner loop controllers control the positions of the carts, directly maintain the balancing, and provide the motion control for the carts as they move to target locations. The outer loop coordinators provide higher level control to the inner loop controllers by sending target locations to them.

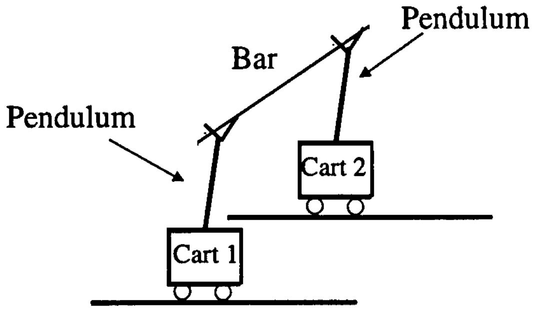


Figure 1: The Coordinated Demonstration System

The objective of the demonstration system is to show that the Simplex system will keep the pendulums and rod balanced while maintaining coordinated motion of the carts, even in the presence of failures. Especially important is the ability to maintain functionality in the event of communication failures.

The Simplex architecture [Sha 96, Simplex 98, STR 98] is a framework for system integration and evolution that supports the online upgrade of software intensive systems and provides fault tolerance in the event of an error in an upgrade. From the perspective of application developers, Simplex is a collection of design principles and real time process management, reliable communication, and fault tolerant facilities.

Simplex provides fault tolerance through redundant variants of components. A typical implementation of the Simplex architecture, for example the outer loop coordination in the coordinated demonstration system used in this case study, consists of a module management unit (decision unit in the case study), one or more application variants (experimental and baseline coordinators in the case study) and a safety variant (safety coordinator in the case study). These outer loop modules are shown in Figure 2.

The decision unit supports the dynamic upgrade operation and responds to faults in the application variants (the experimental and baseline coordinators), switching control as needed. The safety coordinator is treated as a trusted application and the system switches to the safety coordinator when faults occur. Structurally the three coordinators (safety, baseline and experimental) may all be child processes of the decision unit.

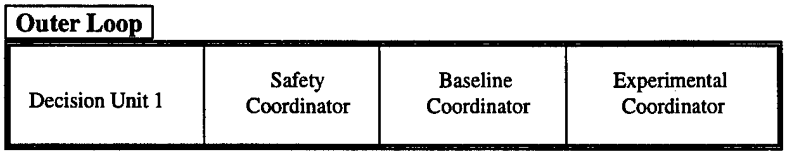


Figure 2: Concurrent Models of the Outer Loop

The experimental coordinator is the upgraded application software and the baseline coordinator is (proven) application software that is being upgraded. The system reverts to the safety coordinator when faults occur. As described earlier, the safety coordinator is the trusted 'core' of the system and encompasses the functionality required to take corrective actions.

An excerpt from the coordinated demonstration system specification is presented in Appendix C. The complete specification [Seto 97] was used to model the coordinated demonstration system.

3 Modeling with Statecharts

Statecharts [Harel 87] provide a means of graphically representing system states and transition relations among those states. They are useful in the design and specification of complex discrete event systems, such as digital control systems, computer-based real time systems, and communication protocols. Statecharts extend the conventional state-machine (state transition) diagrams, with three important elements, the notion of hierarchy, concurrency, and communication. By using the statechart notation, a system being modeled can be represented modularly and refined in stages.

It is not always necessary to represent a system first as a statechart model and then convert it into an SMV model. SMV models can be created directly from the system specifications or transition tables. In the case study, the statechart model of the system was built first to provide a systematic basis for learning the system. It was also important to notice that the statechart approach was very useful in the interaction with the designer, as a graphical depiction of the system was superior to textual depiction during discussions.

3.1 Statechart Notation

A detailed set of statechart notations can be found in [Harel 87]. Figure 3.1 summaries the notation used for the case study which closely follows the Harel convention. States are represented by ovals, and transitions are represented by lines drawn between the states. The enabling of the transitions are governed by transition guards as shown in Figure 3.

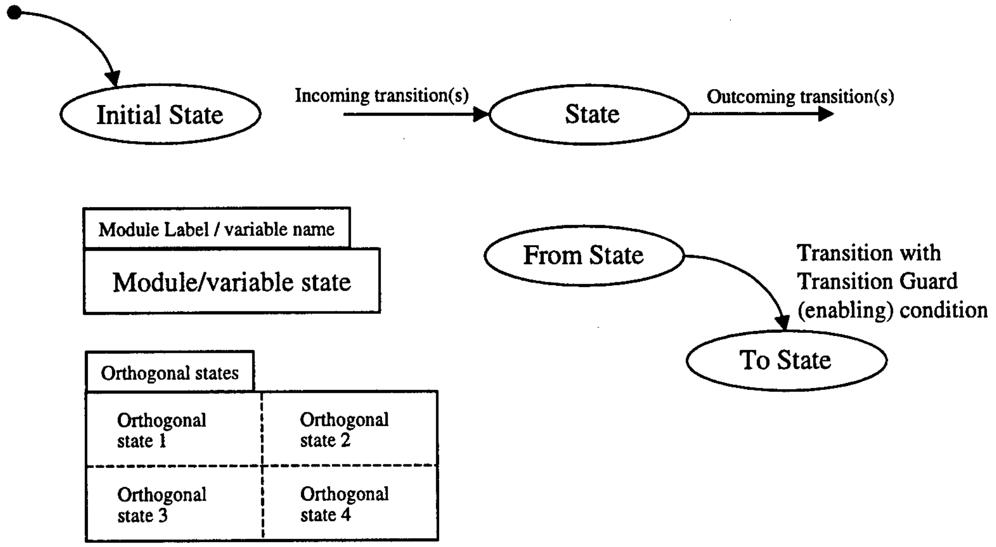

Initial State
Incoming transition(s)
State
Outcoming transition(s)
Module Label / variable name
Module/variable state
From State
Transition with
Transition Guard
(enabling) condition
To State
Orthogonal states
Orthogonal state 1
Orthogonal state 2
Orthogonal state 3
Orthogonal state 4

Figure 3: State Notation Used in the Case Study

3.2 Creating the Statechart Model

The modeling process using statecharts involved the following steps.

1. The first step was confirming the applicability of the statechart approach to the given problem. The system should be easily described as a set of states that capture the important aspects of that system. This implies that it is important to keep in mind what behaviors of the system are most important in meeting the critical system requirements, what performance or properties are being investigated or sought in the modeling effort, and what aspects of the system are important to understand.

   In this study, the fault transition model of the system was the focus of interest. Hence the outer loop controller and its response to error conditions were chosen to be modeled. The inner loop controller was not modeled.

2. The next step is to decompose the system into logical partitions or modules. The outer loop controller can be partitioned into the decision module, safety coordinator, baseline coordinator and experimental coordinator, as shown in Figure 2. These modules form concurrent modules. That is, each of these modules has one active state and the outer loop controller can be described as an ordered n-tuple of these states.

   It is important to recognize that this step is iterative and the further processes of modeling will identify the other concurrent modules (such as the Communication and Pendulum states in the case study).

3. Having identified the concurrent modules, the next step is to refine these modules further into more detailed substates. The process of refinement can be carried out through several levels. The number of levels is generally dictated by the set of properties the user wants to investigate or understand.

   In the case study, two levels of refinement were found to be adequate to discuss the fault-behavioral properties of the system. For example, the overall system was represented as having three states, Initial, Independent, and Coordinated. The Coordinated state was further refined as having modules Decision unit, and the three coordinators. Each of the coordinators were further refined into their respective composite states, such as Alive, Disabled, Terminated, Timing Performance and Position Performance. The refined statechart for the Experimental coordinator is shown in Figure 4.

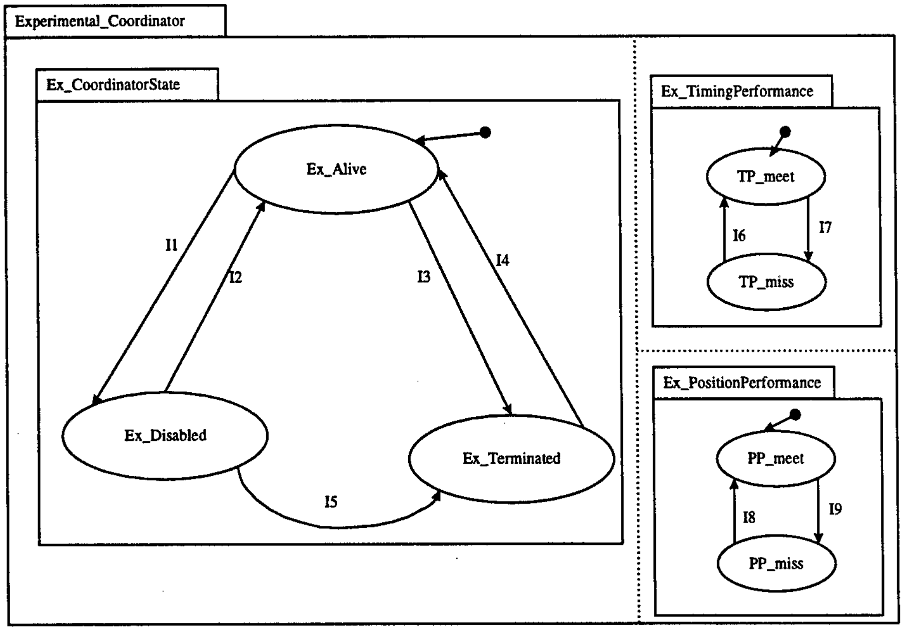


Figure 4: Refined Experimental Coordinator Statechart

4. The next step is to identify the initial system state. The initial states for the system were, Initial (in Overall system module), Experimental Active (in Decision module), Transport (in Safety coordinator module) and Alive (in the other coordinator modules). Fault representation states were initialized to the no-fault state. The system error states, the Communication, Pendulum and Alignment are concurrent states that were required to specify the system completely. These states were concurrent with respect to both Carts and hence they were added as Common States in the Coordinated_demo module.
5. Having identified all the states and the initial states, the transitions among the states and the conditions that govern these transitions are to be identified. There are several ways of doing this. If the state diagrams are not cluttered and the transition relations are not very long and complicated, then they can be captured in the graphical representation itself, with some associated text describing the predicates that govern the transitions. The predicates are guards on the transition.
6. At this point, it is also important to review the given specification to see if any states or transitions are omitted, to identify any transitions that have been added in the module by the modeler's choice and not specified in the document, and to ensure that all diagrams are consistent in naming and description style. Meeting with the designer at this stage was found to be advantageous. In the case study, this step helped to iterate and refine the model and clarify ambiguities in the specification. It also helped in clarifying ambiguous phrases such as "item1, item2, etc."

The result of this process is a graphical statechart representation of the system. In the case study, just going through the modeling process itself uncovered ambiguities in the specifications and acted as a vehicle to unambiguously convey the modeler's understanding of the system to the designer and obtain clarifications. The complete statechart representation of the case study can be found in Appendix B.

4 Modeling Checking with the Symbolic Model Verifier (SMV)

The Symbolic Model Verifier [McMillan 92, SMV 98] provides a means of algorithmically representing a system as a set of states and guarded transitions. It allows for the specification of the properties of the system in Computational Temporal Logic (CTL) notation and assists in checking these properties against the state machine representation. Hence, SMV is a tool for checking that finite state systems satisfy computational tree logic specifications. It uses the Ordered Binary Decision Diagram (OBDD) based symbolic model checking algorithm [Bryant 86].

4.1 Model Checking Steps

Generally model checking involves the automated checking of a model against the expected (desired) properties of a system. In SMV, models are checked against claims and the output of the SMV tool is a confirmation of the claim or a non-confirmation with counter example information. This process is shown in Figure 5.

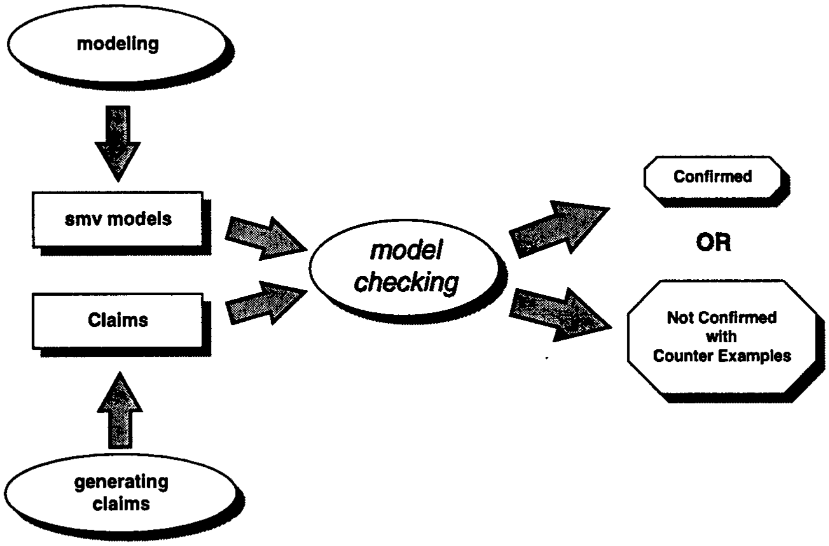

modeling
smv models
Claims
generating claims
model checking
Confirmed
OR
Not Confirmed with Counter Examples

Figure 5: The SMV Model Checking Process

The process of model checking generally consists of the following main tasks:

- **Modeling:** Converting the system specification or design into a formalism accepted by the model checker. For the SMV model checker, models are expressed in the SMV language.
- **Generating claims** about system properties: Before starting to check the model, it is necessary to state what properties the system must satisfy. These properties, claims, are established based upon the specification, and are expressed in SMV using computational tree logic (CTL).
- **Model Checking** the model with respect to the claims: This is an automated process in SMV. The SMV tool, builds a representation of the system state space and checks the claims made, with respect to the model. In doing so, there are several possible outcomes.
  - The claim could be true. In this case, SMV announces that the claim holds.
  - The claim could be false. In this case, SMV announces that the claim does not hold and in most cases provides a counter example (trace) against the specified claim.

It is important to note that model checking only checks the model of the system. For example, when SMV declares a claim as 'true' or 'false', it means that the claim holds or fails to hold with respect to the system model. It does not verify that the model accurately represents the system.

While model checking, the SMV tool may run out of memory or may abort its execution due to system time limitation. In these cases, it is possible to modify control parameters to reduce the execution resources required to check the model. It may also be necessary to simplify the model (e.g., by using a different level of modeling abstraction). These limitations are dependent on both the inherent complexities of the model and the computing resources available.

4.2 SMV Modeling Notation:

The language of SMV is used to describe complex finite state systems. The details of the SMV notation can be found in McMillan [92], but the important features of the language are summarized in the following list.

Modules: The system can be decomposed into modules. Individual modules can be instantiated multiple times and modules can reference variables that are declared in other modules.

Module execution: The assignment statements in SMV are executed simultaneously and in parallel. This is used in cases where system modules execute in a common execution step (e.g., coordinated through a common hardware system clock). If the system modules can interleave in their execution steps, then the keyword process can be used to model asynchronous interleaving among modules.

Non determinism: The state transitions in the system model can be deterministic or non-deterministic. Non determinism can be used, for example, to describe some aspects that are hidden design decisions or allow for deferred decisions.

VAR: This is the variable declaration.

MODULE main: This identifies the main module where the program starts executing.

MODULE: A module is a self-contained set of states and transitions.

ASSIGN: This designates the assignment area, where both the initial and next values are assigned.

INIT: This designates the initial value.

NEXT: This designates the next value, generally predicated on some other states or conditions.

CASE: This is the equivalent of Switch (branching) conditions in the programming language C.

4.3 Creating the SMV Model

This section presents the steps used in creating the SMV model for the case study. Included in the description of each step are comments about the specific process issues and engineering tradeoffs involved.

4.3.1 Learning SMV

Since the modeler was unfamiliar with the SMV approach, the initial step in creating the SMV model for this study involved learning the SMV modeling technique and notation. This was accomplished by first looking at a small example, specifically the ready-busy example shown in Figure 6, and reviewing the SMV literature. The ready-busy example is a two-state system.

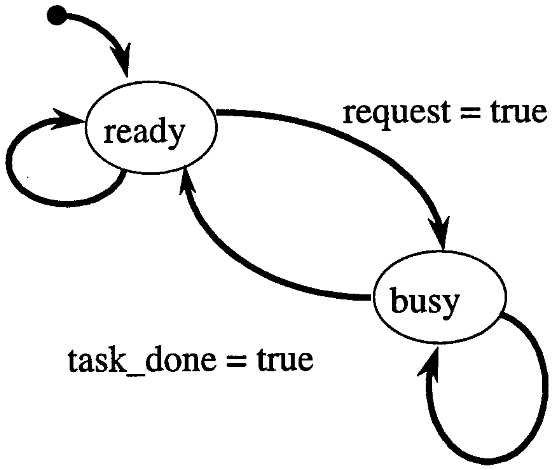


Figure 6: Ready-Busy Example

Figure 7 shows the SMV code for the ready-busy example. The 'main' MODULE consists of the variables 'request', task_done, and 'state'. The variables 'request' and 'task_done' are Boolean and can take only one of two values: true or false. The variable 'state' is of type enumeration that can assume any one of the two values ready or busy. The initial value of state is ready. If the system is in ready state and request is true, the system goes to state = busy. If the system is in busy state and task_done is true, the system goes to state = ready. Otherwise, the system state is not changed.

```
MODULE main
VAR
request : boolean;
task_done : boolean;
state : {ready, busy};
ASSIGN
init(state) := ready;
next(state) := case
state = ready & request : busy;
state = busy & task_done : ready;
1 : state ;
esac;
```

Figure 7: SMV Model of the Ready-Busy Example

4.3.2 Variables

The next step is to declare the variables corresponding to the states in the statechart. Two points must be borne in mind in this process. The variable names should be meaningful and consistent. The second point is that variable types (Boolean or enumerated or array) and the scope of variables (the modules where the variable is available for change) need to be determined carefully. If a logical state is to be represented, a Boolean variable is the preferred type. If the variable takes only a few values, then enumerated types are suitable. The other types are to be used sparingly, as they very quickly contribute to the state explosion problem. Any unused or redundant variables should be deleted in this step.

4.3.3 Model Refinement

SMV follows the same refinement principles as those used in statecharts. For example, Cart1, which is a module, is declared as a variable. This Cart1 module is then made up of other variables and modules. Thus, the composition is hierarchical where each statechart module is converted into an SMV Module. Each state within a module becomes an enumerated (or Boolean) value for the variable.

In the case study the experimental coordinator module which includes the Coordinator state, timing performance, and position performance is converted into an SMV module. The statechart representation for experimental coordinator module is shown in Figure 4. The SMV representation is shown in Figure 8 where each concurrent state machine in Figure 4 is modeled as a variable. In the SMV model the transitions among the states are represented as boolean predicates that control (enable or disable) the transitions. The initial state of a module is represented as init(state) and the subsequent state, the one that occurs at the next execution, is represented as next(state). These state transitions are governed by predicates typically encapsulated in a case statement. As shown in Figure 8 on the third line from the bottom, the next state for the experimental coordinator, when both the predicates (state = terminated) and (User.Ex_cked = create) are true, is the alive state.

```
- the Experimental unit
MODULE Experiment(User,S_obs)
VAR
 state : {alive,terminated,disabled};
 time  : {meet,miss};
 perf  : {meet,miss};

ASSIGN
 init(state) := alive ;
 next(state) :=
 case
   ((state = alive) & ((time =miss) | (perf = miss) | (S_obs.status=unsafe))) | ((state = alive) & (User.Ex_cked = disable)) : disabled;
   ((state = alive) & (User.Ex_cked = kill))     : terminated;
   (state = disabled) & (User.Ex_cked = kill)    : terminated;
   (state = disabled) & (User.Ex_cked = enable)  : alive;
   (state = terminated) & (User.Ex_cked = create) : alive;
   1: state;
 esac;
```

Comments: User is the reference to the user commands module that gives the user initiated commands such as kill and create. S_obs is the reference to the fault status of the cart. It has a variable status which takes the values safe or unsafe.

Figure 8: The SMV Representation of the Experimental Coordinator

4.3.4 Iterations of the Model

As with the statechart model, the SMV modeling process required a few iterations. The modeling discussions with the designer, helped to clarify some implicit assumptions in the specifications and correct minor misconceptions in the initial versions. For example, the module Safety coordinator was initially modeled as having Transport state and error indicator modules. The initial impression in reading the specification was that the Communication, Pendulum, and Align states were equal priority error indicator states. But the modeling discussions showed that these states are not error indicator states, but rather error handing states, where actions were initiated to respond to the error. Hence, the Safety coordinator was modified to have states Transport, and error handling states Handle_Comm, Handle_Pen and Handle_align, as shown in Figure 9. With this modification, the implicit priority and hierarchy in handling the three errors was also clearly brought out, with communication errors given the highest priority and alignment errors given the least priority.

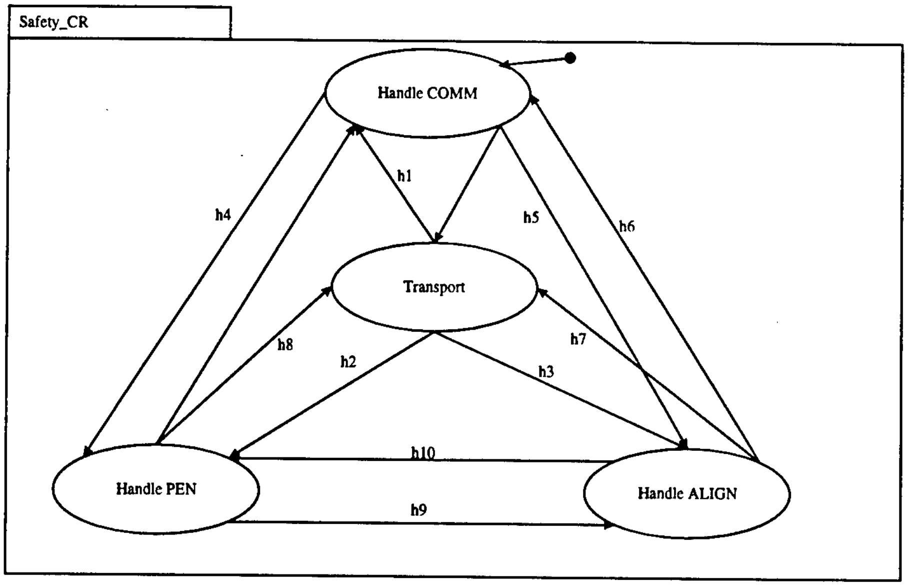


Figure 9: Safety Coordinator States

4.3.5 Syntax Checking

Syntax checking is the next step in the modeling process. The syntax and consistency of the model can be checked using the SMV tool, without making a temporal logic claim. However, this checking does not guarantee that the model is correct, but rather identifies basic syntax errors and ensures that the model can be processed by the error checker.

4.4 Checking the Claims

In SMV, claims are written as CTL formulae. Claims are included in the SMV input at the end of the Module main, under the keyword SPEC. The CTL operators used in SMV are combinations of A (for all), F (eventually), G (globally), E (there exists), and X (in the next state). These operators can be combined with Boolean operators of first order logic (such as NOT, AND, OR, and IMPLIES).

As stated before, the checking of claims in SMV is an automated process. SMV builds the OBDD representation of the modeled system and attempts to check the claims. If it encounters an error during the checking, it provides a counterexample so that the user can use this trace to identify any inconsistencies. In some cases, especially with EF (there exists a path) specifications, SMV cannot give specific counterexamples. This is because, to give a counterexample that a path does not exist, it must enumerate all possible paths and show that the required path does not exist.

A few example claims used in the case study and their results are discussed here. In each subsection that follows the claim is listed along with its meaning, the results of the SMV tool checking of the claim, and an interpretation of the results.

4.4.1 Claim 1

The claim for the first example is shown below:

```
AG (( G_COMM = 0) -> (AF( cart1.sf.state = han_com)))
```

Meaning

This claim states that, if there is a communication failure the system will in all cases (always) handle the fault. The explicit translation of the notation for the claim is that for all computation paths globally, if the communication fails, then the system will eventually get to a state of handling the communication fault.

SMV Result

specification `AG (( G_COMM = 0) -> (AF( cart1.sf.state = han_com)))` is true

Interpretation of the Result

SMV confirms that the claim holds. Consequently, in any system state, if the communication fails the system will handle the communication error.

4.4.2 Claim 2

The claim for this example is shown below.

```
AF (( G_COMM = 0 &  G_PEN = 0) -> (AF( cart1.sf.state = han_pen)))
```

Meaning

This claim states that if the communication has failed and there is a pendulum fault at the same time, the system will always handle the pendulum fault. The explicit transition of the notation for the claim is that for all computation paths, if the communication has failed and the pendulum is in the fault state, then eventually the system will always get to the pendulum fault handling state.

SMV Result

This claim produced a lengthy trace, as shown in Table 1. The trace is read down each column proceeding from the leftmost column to the right.

```
% smv -c 16381 -k 16381 -f -r CoorSystem.SMV
-- specification AF (G_COMM = 0 & G_PEN = 0 -> AF cart1.s... is false
-- as demonstrated by the following execution sequence
state 1.1:
sys = initial
target = 1
coor_on = 0
G_COMM = 0
G_PEN = 0
C1_sf.COMM = 0
C1_sf.PEN = 0
C1_sf.ALIGN = 0
C1_sf.status = safe
C2_sf.COMM = 0
C2_sf.PEN = 0
C2_sf.ALIGN = 0
C2_sf.status = safe
cart1.User.Bl_cked = disable
cart1.User.Ex_cked = disable
cart1.sf.state = transport
cart1.Ex.state = alive
cart1.Ex.time = miss
cart1.Ex.perf = miss
cart1.Bl.state = alive
cart1.Bl.time = miss
cart1.Bl.perf = miss
cart1.DU.D_select = exp
cart2.User.Bl_cked = disable
cart2.User.Ex_cked = disable
cart2.sf.state = transport
cart2.Ex.state = alive
cart2.Ex.time = miss
cart2.Ex.perf = miss
cart2.Bl.state = alive
cart2.Bl.time = miss
cart2.Bl.perf = miss
cart2.DU.D_select = exp
state 1.2:
sys = independent
C1_sf.status = unsafe
C2_sf.status = unsafe
cart1.sf.state = han_com
cart1.Ex.state = disabled
cart1.Bl.state = disabled
cart2.sf.state = han_com
cart2.Ex.state = disabled
cart2.Bl.state = disabled
-- loop starts here --
state 1.3:
cart1.DU.D_select = safety
cart2.DU.D_select = safety
state 1.4:
resources used:
user time: 0.86 s,
system time: 0.06 s
BDD nodes allocated: 10542
```

Table 1: SMV Trace for Claim 2

Interpretation of the Results

The claim was declared false and a counterexample was provided; i.e., there is a case where the pendulum fault would not be handled. This situation occurs, as noted in state 1.1 and state 1.2 in Table 1, in the case that there is a communication failure (`G_COMM = 0`) and the system never recovers from that failure. In this situation, no other errors can be handled because handling a communication fault is the highest priority. Hence, the system is stuck in the communication fault handling state, as expected based upon the priority fault-handling scheme.

4.4.3 Claim 3:

The claim for this example is shown below

```
AF (( G_COMM = 0 &  G_PEN = 0) -> (EF( cart1.sf.state = han_pen)))
```

Meaning

This claim is a modification of claim 2, which explores whether the pendulum fault is handled, not always but rather will be handled under some circumstances. The explicit translation of the notation for the claim is that for all computation paths, if the communication has failed and the pendulum is in fault, then eventually there exists a path by which the system can get to the state of handling the pendulum fault. In particular for this claim the second AF of claim 2 is replaced by EF. The EF states that 'there exists a path' in contrast to AF which states 'for all paths.'

SMV Result

specification `AF (G_COMM = 0 & G_PEN = 0 -> EF cart1.s...` is true

resources used:
user time: 0.72 s
system time: 0.11 s
BDD nodes allocated: 10032

Interpretation of the Results

For this claim when the communication fails, the system handles the fault and demonstrates that the system can handle the pendulum fault if the communication is restored. The success of the claim indicates that there is a path that will eventually take the system to the handling the pendulum fault state.

5 Process Metrics and Observations

Time logging was done during the course of this case study. This consisted of the time taken to learn the specification, learn the SMV tool, model the system, incorporate changes, and generate and check claims. These results are summarized as percentages in Table 2 and each of the activities is described in more detail in subsequent subsections. The information includes key characteristics of the effort, description of the work completed, and general comments on the activity.

The percent of effort in each category is presented here. Through the capture of data from future case studies we hope to get statistically significant metrics that can be used to make predictions about the level of effort required to implement model-based verification practices in specific problem types and provide a comparative norm across projects.

Activity	**Effort Percent**	**Comments**
Learning the System	33%	37% specific meetings and 63% in reviewing specifications and related material
Learning SMV	15%	This involved review of manuals, completing example problems, and consultation with experts.
Modeling the System	38%	15% in meetings, 57% statechart modeling and 28% hours in SMV modeling
Changes	4%	50% in meetings and 50% in model changes and checking
Model Checking	10%	This does not include time for someone to learn CTL.
Total =	100%	This represents all of the time spent on the effort.

Table 2: Percent Time in Each Major Activity

5.1 Learning the System

The objective of this activity was to understand and clarify the details of the system as represented in the specification.

Description

The specifications were carefully read and reviewed. A series of meetings were held with the designer to clarify several points in the specifications. These consisted of a series of five 60-to-90-minute meetings spread over 2.5 months. Additional background material, related to the system specification was also reviewed (e.g., notes about the Simplex architecture).

Observations and Comments

The specification document was very concise consisting of text supported formal notation, principally set notation. Interpretations of the symbols were consistent across the document.

The individual doing the modeling had no prior knowledge of the system.

The meetings with the designer were critical in uncovering some subtleties in the notation used in the specification.

5.2 Learning the SMV Modeling Language and Tool

The objective of this activity was to learn the SMV modeling language and how to use the SMV tool.

Description

All of the documents on SMV listed in the references section were reviewed and small sample problems in SMV were completed. A few discussions with the SMV research group within the School of Computer Science at Carnegie Mellon were also held.

Observations and Comments

The modeler had no previous experience with the tool other than a limited number of prior interactions with the SMV research group. These interactions with the SMV research staff were impromptu and involved short conversations during encounters in the hallway or at departmental seminars, colloquia, or social events.

The SMV modeling language is similar to the C programming language. It is straightforward and was easy to learn. It is expected that learning the tool will be straightforward for any individual with C programming experience and some knowledge of state machines.

5.3 Modeling the System

The objective of this activity was to generate a model of the system using statecharts and later SMV.

Description

With the understanding that the modeler achieved after reading the system specifications, the system was incrementally modeled in statechart notation and meetings with the designer were held. These meetings further refined the models. When it was felt that the model reached a fair level of stability (not many corrections in a meeting), the statechart model of the system was converted into SMV.

Observations and Comments

The modeler had completed a formal modeling course as part of the Master of Software Engineering program at Carnegie Mellon [Garlan 98, MSE 98].

Modeling of the system was a very iterative process. Generating the statemachine model took time but the conversion to an SMV model was straight forward, almost seamless.

Because the study was not a full time effort over the time period for the case study, separate periods of work sometimes involved going back and re-reading and reviewing work already completed and relevant artifacts before continuing the modeling process. The modeling required multiple modeling sessions including 6 versions of statecharts and 3 versions of SMV code.

5.4 Making Changes to the Models

The objective of this activity was to update the models (statecharts and SMV), based upon review and comment by the Simplex designed.

Description

Two changes were to be incorporated. The first change was to modify the representation of the safety coordinator. The initial impression of the safety coordinator was that it acts as an error announcer. But during the discussions it turned out that the safety coordinator actually handles the errors such as communication, pendulum and alignment. The second change was that the communication and pendulum faults were common to both the carts but an alignment fault was not. This had to be reflected in the model.

Observations and Comments

The models were intentionally kept modular, facilitating changes. The first change of incorporating error-handling states instead of the error announcing states was straightforward. Because of the modular nature of the modeling (both in the statechart and the SMV notations), the change was localized. With the incorporation of the error handling states, the required priority in the error handling with communication error handling given the top most priority and the alignment error handling given the least priority was also incorporated.

The second change of separating the hitherto common error representation states (communication, pendulum and alignment) into the categories of 'common to both carts' or 'specific to each cart' was not so straightforward. This change would allow a situation where one cart could be in 'unsafe' condition because of an alignment fault whereas the other cart could be in 'safe' condition. For example, alignment was no longer common to both carts, but was now an autonomous property of each cart. This led to some major changes in two modules: Common states and Safety coordinator, shown in Figure B1. (The side effect of this was to also change some of the claims that were written. This was needed, as new variables were introduced during this modification.)

5.5 Generating and Checking Claims

The objective of this activity was to identify a set of interesting properties about the system, express them in CTL notation, and check the model against those claims.

Description

During meetings with the designer, a few important properties about the system of particular interest to the designer were identified. These were converted into CTL claims and the SMV tool was used to check these claims. When the variables used in the model changed, some of the claims were changed and model checking was repeated.

Observations and Comments

The modeler had a formal modeling course as part of the Master of Software Engineering Program and consequently was very familiar with temporal logic. In addition to general knowledge of temporal logic, expertise in Computational Tree Logic (CTL) is required. CTL is a very expressive notation and extreme care is required in formulating the claims.

6 Summary

Using the Symbolic Model verifier (SMV) as the principal modeling approach, the goal of this study was to explore practice and process issues involved in the modeling and model checking of the Simplex coordinated demonstration system. As a precursor to the SMV model, a statechart model was created. Both the technical design specification and discussions with the principal designer were used as the foundation for generating the statechart and SMV models as well as for establishing claims about the system. Throughout the modeling effort relevant engineering decisions, issues, and problem were recorded and the time required for each of the major activities was logged. This section presents some general observations and a summary of this work.

6.1 Observations on the Modeling Effort

Modeling the system using the statechart notation was found to be very useful for three reasons. First, the graphical notation was simple to use, helpful in learning the system, and effective in facilitating communications during meetings with the designer. Second, the conversion of the statechart model to the SMV model was very straightforward, with the structure providing an easy guide to creating assignment, transitions, and states within the SMV model. Third, the modular and hierarchical graphical structure helped to facilitate changes to the model.

Learning the SMV tool was straight forward, as the notations were similar to common programming languages like C and the modeler was an experienced C programmer. While converting the statechart representation into SMV was straight forward, stating CTL claims about the system and verifying that a claim represents what is actually meant, required expertise and experience. Stating the claims was one of the more difficult technical challenges of this study. Once the claims were stated, checking of the claim was automatically done by the tool. The counterexamples provided for inconsistent claims were very helpful in further analysis and debugging of the system (model). The time for the SMV tool to complete the model checking was on the order of seconds.

The specification of the coordinated demonstration system was, in large part, represented in a formal notation and no significant errors (e.g. deadlocks or starvation) were found. However, the case study identified some minor ambiguities in the specification, the most noteworthy being in the representation of fault states. It was not clear that the states communication fault and pendulum fault were common to both carts, especially since in contrast to this commonality, the alignment fault was specific to each of the individual carts.

The meetings with the designer were very useful in providing clarification and benefit to the designer, prompting him or her to look at different perspectives. In the general, application of this approach involving meetings with designer, clients, and other stake holders should help to ensure that the model represents the system faithfully.

Modeling is not a new process to engineers. Software engineers have their own mental models before developing software. What this case study reiterates is that there is lot to be gained if the models are formal. During this effort it was realized that the formal modeling process itself, independent of formal model checking, is capable of uncovering inconsistencies and ambiguities in a software specification. It was evident that the rigor and discipline of the notation channeled the modeler to ask detailed questions and seek clarifications. It is expected that these characteristics will be present in the general application of these techniques.

Models can enable software engineers to reason about the properties of a system and help to establish more formal syntax and semantic definitions that can be used to represent the system unambiguously. Additional insight into the system, especially looking at complexity that cannot be unraveled through manual review, can be gained through the automated checking of the formal model. As others have noted, the checking of the model with respect to the claims, is an additional benefit and can be used as required on a case by case basis [Clarke 96, Jackson 96].

6.2 Observations on the Practice

It is noteworthy that the rate of pages reviewed per hour for this case study was less than that noted by Gilb [93], a rate of 0.16 pages per hour in this study versus 0.5 to 1.5 or more pages for a conventional review. Others cite substantially higher rates (e.g., Jones presents a range of 12 to 50 pages per hour depending on the type of specification and whether the time is in preparation or meetings [Jones 91]). This result is not especially discouraging, considering that the specification involved extensive formal notation and an inexperience modeler, this was the first real problem the modeler analyzed. In addition, these techniques have been shown to identify errors that have eluded detection by conventional techniques [Clarke 95, Clarke 96b, Fujita 96, Raimi 97].

In this effort two models were created; a statechart and a SMV model. In applying the SMV technique, a more experienced SMV modeler would likely not create a statechart model of the system. In addition, there was extra time spent in pedagogical activities that increased the total time required to complete the modeling effort and reduced the pages per hour rate. For example, meetings held during the modeling effort and related activities would often involve achieving an understanding how to use a particular technique or capability of the modeling system rather than focusing on clarifying specifics of the model itself. These activities would not be a routine part of a practical application of these techniques, where more experienced modelers would be involved. While the explicit time required to learn the tool was factored out of the data used to calculate the page per hour rate, there still remained the exact time related to pedagogical activities, which proved too difficult to extract from the raw data.

Since there is a fixed overhead to starting any modeling activity (e.g., setting up the modeling environment), this overhead may bias the results to lower page per hour rates in the case of small specifications. Consequently, it is the opinion of the authors that the low rate observed in this study is not indicative of that for more experienced modeling engineers working on larger problems. Future investigations involving larger case studies and real-world projects will help to provide additional data to evaluate this opinion.

6.3 Observations on Applicability

This study involved the analysis of a specification that included both requirements and design. While only the specification was used by the modeler for the analysis, an implementation of the system was available. The code for the implementation was used by the designer to resolve a few issues that were raised in the study. Based upon this study, there does not appear to be any impediment evidenced that would preclude the application of the modeling approach to any phase of the software development life cycle, including requirements gathering, design, and maintenance. For example, the modeling approach could be implemented as part of a peer review process during a maintenance upgrade. One possible approach is to have one or two members of a review team who are proficient in modeling use the approach as part of their review activities.

The process of modeling may be an effective way of allowing an engineer, new to a project to learn more about the system. Instead of explaining the entire system to the new individual, they can be assigned the responsibility of creating a formal model of the system. This would provide a framework for interacting with other designers and stakeholders, as appropriate, and establishing more detailed understanding of the system by generating and checking claims.

Even if the low rates of pages per hour observed in this study are confirmed in subsequent investigation, these techniques may find cost-effective application in the selective analysis of the critical aspects of a system, where extra efforts to ensure reduced error rates are warranted. For higher assurance applications, where error elimination is a principal concern, the higher costs associated with these techniques will likely be acceptable. In any event, it is premature to make definitive judgements based upon this limited investigation.

6.4 Future Work

This study can be viewed as one preliminary step in fostering the transition of model-based verification techniques and the lightweight use of formal models for software development generally. Additional investigations that capture the technical and practice issues in applying model-based techniques will be necessary to establish their viability in broader software engineering practice.

7 References

[Bryant 86]	Bryant, R. E. "Graph-Based Algorithms for Boolean Function Manipulation." *IEEE Transactions on Computers C-35*, 8 (August 1986): 677-691.
[Clarke 86]	Clarke, E.; Emerson, E.A.; & Sistla, A.P. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications." *ACM Transcripts on Program Language Systems 8*, 2 (1986): 244-263
[Clarke 95]	Clarke, Edmund M., et. al. "Verification of the Futurebus+ Cache Coherence Protocol." *Formal Methods in System Design 6*, 2 (March 1995): 217-232.
[Clarke 96a]	Clarke, E. & Kurshan, R. "Computer Aided Verification." *IEEE Spectrum 33*, 6 (June 1996): 61-67.
[Clarke 96b]	Clarke, E. M. & Wing, Jeannette. "Formal Methods: State of the Art and Future Directions." *ACM Computing Surveys 28*, 4 (December 1996): 626-643. Also CMU-CS-96-178.
[Fujita 96]	Fujita, M. "Debugging a Communications Chip," *IEEE Spectrum 33*, 6 (June 1996): 64.
[Garlan 98]	Garlan, D.; Gluch, D.; & Tomayko, J. "Agents of Change: Educating Software Engineering Leaders." *IEEE Computer 30*, 11 (November 1997): 59-65.
[Gilb 93]	Gilb, T. & Graham, D. *Software Inspection*. Susannah Finz, ed. Wokingham, England: Addison-Wesley, 1993.
[Gluch 98]	Gluch, D. & Weinstock, C. *Model-Based Verification: A Technology for Dependable System Upgrade* (CMU/SEI-98-TR-009, ADA 354756). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1998.

[Harel 87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming 8*, 3 (June 1987): 231-274.

[Jackson 95] Jackson, M. *Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices.* New York: ACM Press, Addison-Wesley, 1995.

[Jackson 96] Jackson, Daniel & Wing, Jeannette. "An Invitation To Formal Methods: Lightweight Formal Methods." *IEEE Computer 29*, 4 (April 1996): 21-22.

[Jackson 98] Jackson, M. "Formal Methods and Traditional Engineering." *Journal of Systems and Software 40*, 3 (March 1998): 191-194

[Jones 91] Jones, C. *Applied Software Measurement: Assuring Productivity and Quality.* New York: McGraw-Hill, 1991.

[McMillan 92] McMillan, K.L. (1992) *Symbolic Model Checking: An Approach to the State Explosion Problem* (CMU-CS-92-131). Pittsburgh, Pa.: Computer Science Department, Carnegie Mellon University, 1992.

[MSE 98] School of Computer Science, Carnegie Mellon University. "Master of Software Engineering [home page]." <http://www.cs.cmu.edu/afs/cs/project/mse/www/>, Wednesday, September 9, 1998; 4:17 P.M. EDT.

[Pnueli 80] Pnueli, A. "A Temporal Logic of Concurrent Programs." *Theorems of Computer Science 13*, 1 (1980): 45-60.

[Raimi 97] Raimi, R. & Lear, J. "Analyzing a PowerPC™ 620 Microprocessor Silicon Failure Using Model Checking," 964-973. *Proceedings of the International Test Conference 1997*, Washington, DC, November 1-6, 1997.

[Seto 97] Seto, D. "Distributed Coordinated Motion of Two Inverted Pendulums," Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, internal communications, 1997.

[Sha 96]	Sha, Lui; Rajkumar, Ragunathan; & Gagliardi, Michael. "Evolving Dependable Systems," 335-346. *Proceedings of 1996 IEEE Aerospace Applications Conference on Reliability and Quality of Design*, Part 1, Aspen, Colo., Feb. 1996. Piscataway, N.J.: IEEE Service Center, 1996.
[Simplex 98]	Altman, Neal. "Simplex Architecture." <http://www.sei.cmu.edu/activities/simplex/simplex_architecture.html>, Wednesday, September 9, 1998, 4:27 P.M.
[SMV 98]	Beregey, Serezin. "Formal Methods – Model Checking." <http://www.cs.cmu.edu/~modelcheck/>, Thursday, September 3, 1998; 11:32 A.M. EDT.
[STR 98]	Weinstock, Charles. "Software Technology Review: Simplex Architecture." <http://www.sei.cmu.edu/str/descriptions/simplex_body.html>, Wednesday, September 9, 1998, 4:32 P.M. EDT.

Appendix A: Glossary

BDD	Binary Decision Diagram
CMU	Carnegie Mellon University
CSP	Communicating Sequential Processes
CTL	Computational Tree Logic
CVT	Continuous Verification and Test
MSE	Master of Software Engineering
OBDD	Ordered Binary Decision Diagrams
O-O	Object-oriented
SCS	School of Computer Science
SEI	Software Engineering Institute
SMV	Symbolic Model Verifier
TL	Temporal Logic
URL	Universal Resource Locator

Appendix B: State Model of the System

This appendix is a compendium of the statecharts for the system.

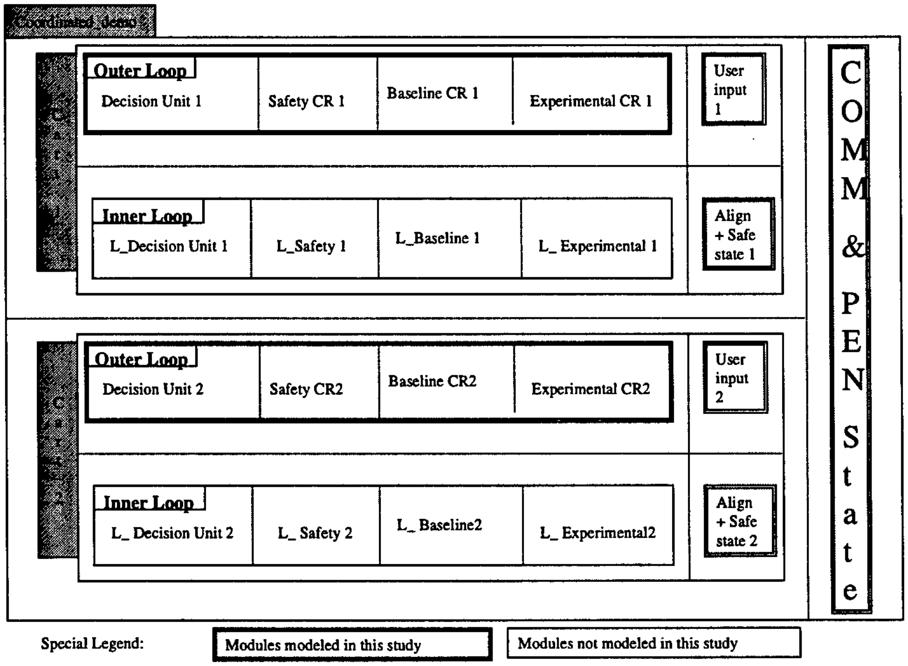

Coordinated_demo
Outer Loop
Decision Unit 1
Safety CR 1
Baseline CR 1
Experimental CR 1
User input 1
Inner Loop
L_Decision Unit 1
L_Safety 1
L_Baseline 1
L_ Experimental 1
Align + Safe state 1
Outer Loop
Decision Unit 2
Safety CR2
Baseline CR2
Experimental CR2
User input 2
Inner Loop
L_ Decision Unit 2
L_ Safety 2
L_ Baseline2
L_ Experimental2
Align + Safe state 2
COMM & PEN State
Special Legend:
Modules modeled in this study
Modules not modeled in this study

Figure B1: Overall Configuration of the System

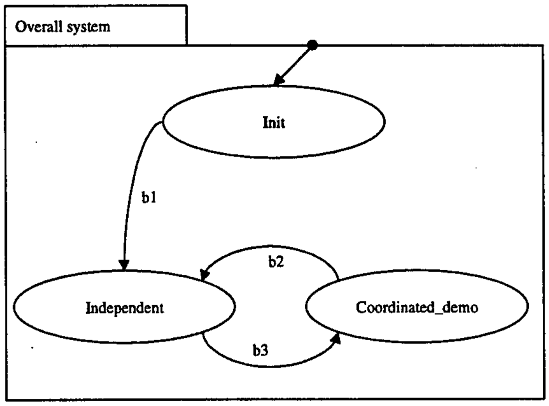

Overall system
Init
b1
b2
Independent
Coordinated_demo
b3

Figure B2: States of the Overall System

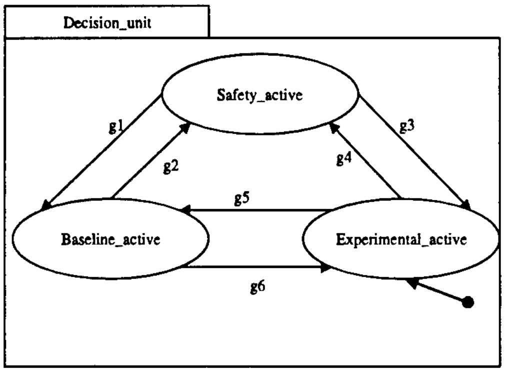

Decision_unit
Safety_active
g1
g2
g3
g4
g5
Baseline_active
Experimental_active
g6

Figure B3: States of the Decision Unit

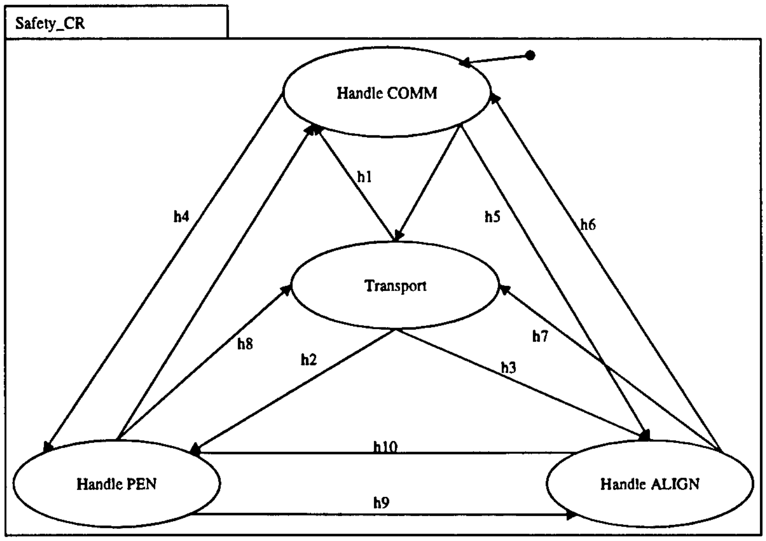

Safety_CR
Handle COMM
h1
h4
h5
h6
Transport
h8
h2
h7
h3
h10
Handle PEN
Handle ALIGN
h9

Figure B4: States of the Safety Controller

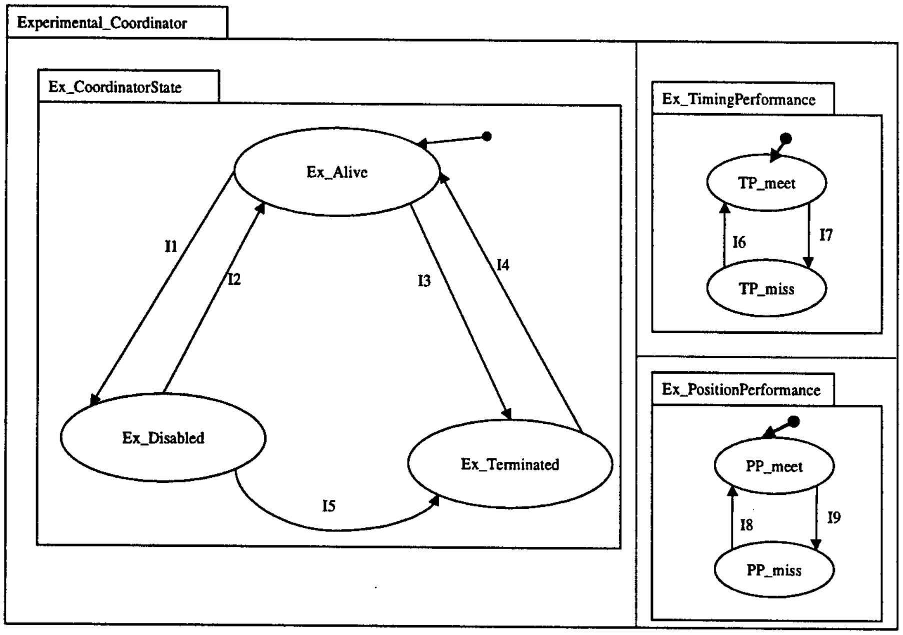

Experimental_Coordinator
Ex_CoordinatorState
Ex_Alive
Ex_Disabled
Ex_Terminated
I1
I2
I3
I4
I5
Ex_TimingPerformance
TP_meet
TP_miss
I6
I7
Ex_PositionPerformance
PP_meet
PP_miss
I8
I9

Figure B5: States of the Experimental Coordinator

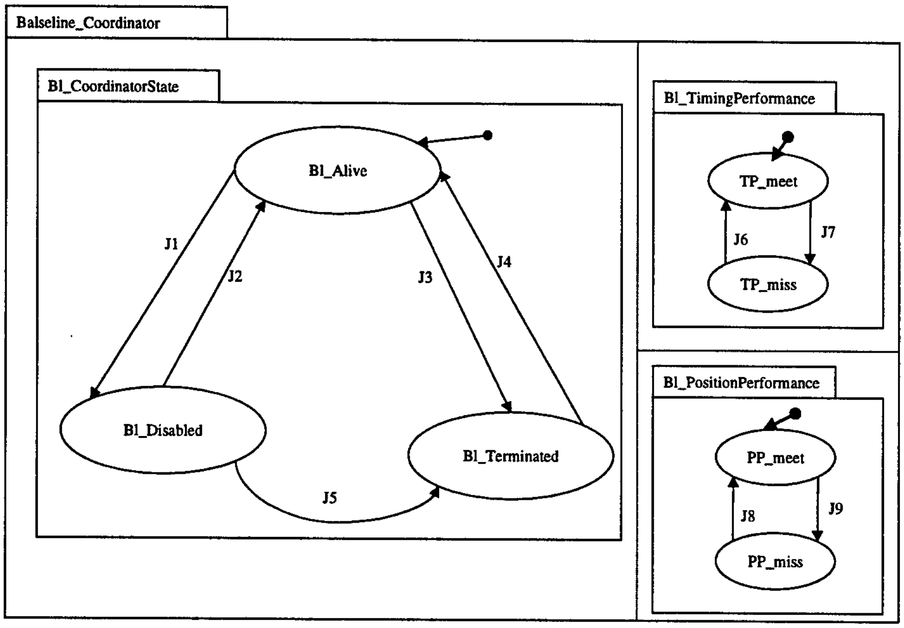
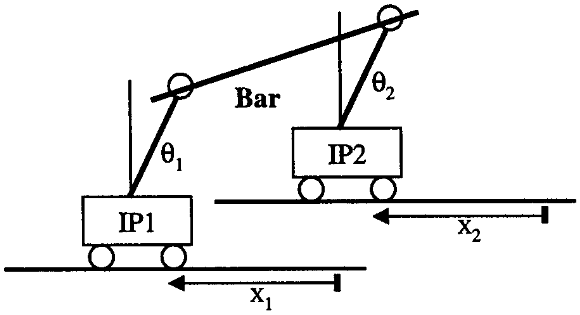

Balseline_Coordinator
Bl_CoordinatorState
Bl_Alive
Bl_Disabled
Bl_Terminated
J1
J2
J3
J4
J5
Bl_TimingPerformance
TP_meet
TP_miss
J6
J7
Bl_PositionPerformance
PP_meet
PP_miss
J8
J9

Figure B6: States of the Baseline Coordinator

Appendix C: An Excerpt of the Specification

An excerpt from the specification used for this study is included in this Appendix to show the level of detail and the notation used by the designer. The entire fourteen page specification was used for the study described in this report.

Distributed Coordinated Motion of Two Inverted Pendulums



Outer Loop Control

The outer loop control deals with the coordination of the motion of two inverted pendulums.

It consists of a safety coordinator, a baseline coordinator, and an experimental coordinator. The coordination is accomplished by generating and adjusting the setpoints of the inner loop controllers.

Specifications

User's Command

1. TARGET - Set the target track position
2. COOR_ON - Turn on the coordination. The pendulum will move coordinately
3. COOR_OFF - Turn off the coordination. The pendulum will move independently.

An Example of Sequential Control Procedure by the User

- Issues the commands TARGET and COOR_OFF to move the pendulums independently to the position where the bar will be engaged.
- After the pendulums align up at the target position, issues the command COOR_ON;

- After the pendulum are coordinately stabilized at the target, engage the bar.
- Issues the commands TARGET and COOR_ON to move the pendulums to a new target position with the bar engaged.

Communication Assumption

Information of the current state of the pendulum and current status of the inner loop controllers need to be exchanged between two trusted computers if COOR_ON is demanded.

Timing Performance of a Coordinator

The set of events which describe coordinator timing performance, *CR_TP* is defined as

$$CR\_TP = \{\mathrm{CR\_MEET}, \mathrm{CR\_MISS}\}$$

where CR_MEET/CR_MISS indicate that the coordinator meets/misses the deadline.

Fault Model

- Communication fault: At least one trusted computer does not get message from the other.
- Pendulum fault: At least one of the pendulums is under the inner loop safety control.
- Alignment fault: the tips of the pendulums are far apart, i.e.,

$$|(x1+l\sin\theta 1) - (x2+l\sin\theta 2)| > dsafe \text{ or } |l\cos\theta 1 - l\cos\theta 2| > hsafe$$

Safety Requirement

- Pendulum Survival: Pendulums do not fall.
- Safety Criterion: Same as single pendulum
- Safety for coordination: Do not drop the bar
- Safety Criterion: The coordination is said *safe* if:
  - Both trusted computers receive message in a specified rate.
  - None of the pendulums is under inner loop safety control.
  - $|(x_1+l\sin\theta 1) - (x2+l\sin\theta 2)| <= dsafe$ and $|l\cos\theta 1 - l\cos\theta 2| <= hsafe$.

Otherwise, it is *unsafe*.

State of Coordination Safety

The set of the states of the coordination, *CR_SAFE,* is defined as

$$CR\_SAFE = \{\mathrm{SAFE}, \mathrm{UNSAFE}\}$$

where SAFE/UNSAFE indicate the coordination is safe/unsafe.

Performance Requirement

The pendulums carry the bar to the target position in a given time interval. Specifically, let MAX_ST_UPDATE and MIN_ST_UPDATE be the maximum and minimum setpoint updates in one outer loop sampling period; Prev_ST and Current_ST be the setpoints in the previous period and current period, and Target be the final track position that the pendulum is required to reach. The coordinator which generates the Current_ST is said *perform* if:

```
Current_ST = Target

     when |Prev_ST - Target| <= MAX_ST_UPDATE;

Prev_ST + MIN_ST_UPDATE <= Current_ST <= Prev_ST + MAX_ST_UPDATE

     if Prev_ST < Target - MAX_ST_UPDATE;

Prev_ST - MAX_ST_UPDATE <= Current_ST <= Prev_ST - MIN_ST_UPDATE

     if Prev_ST > Target + MAX_ST_UPDATE;
```

Otherwise the coordinator is said *non_perform.*

State of Coordinator Performance

The set of the status of a coordinator's performance, *CR_PERF,* is defined as

$$CR\_PERF = \{P, NON\_P\}$$

where P/NON_P indicate that the coordinator is perform/non_perform.

User's Command of Managing a Coordinator

The set of the User's Commands to manage a coordinator, *CR_UC,* is defined as:

$$CR\_UC = \{CR\_C, CR\_K, CR\_E, CR\_D, 0\}$$

where CR_C/CR_K creates/kills the coordinator, CR_E/CR_D enables/disables the output of the coordinator, and 0 means no command.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. **AGENCY USE ONLY** (LEAVE BLANK)	2. **REPORT DATE** November 1998	3. **REPORT TYPE AND DATES COVERED** Final

4. **TITLE AND SUBTITLE**
A Study of Practice Issues in Model-Based Verification Using SMV

5. **FUNDING NUMBERS**
C — F19628-95-C-0003

6. **AUTHOR(S)**
Grama R. Srinivasen
David P. Gluch

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. **PERFORMING ORGANIZATION REPORT NUMBER**
CMU/SEI-98-TR-013

9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
ESC-TR-98-013

11. **SUPPLEMENTARY NOTES**

12.A **DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified/Unlimited, DTIC, NTIS

12.B **DISTRIBUTION CODE**

13. **ABSTRACT** (MAXIMUM 200 WORDS)

This report presents the results of a case study into practice issues involved in using the Symbolic Model Verifier (SMV) for model checking software systems. The case study is of a Simplex implementation—the Simplex coordinated demonstration system for reliable system upgrade. The investigation consisted of generating a system model (using both statechart and SMV notations), specifying claims (expected properties) of the system as temporal logic formulae, and checking those formulae with respect to the SMV model. The various steps involved in the modeling process are described. Examples of the claims, their results, and a description of how the SMV tool analyzed them are detailed. Key engineering decisions made during the modeling process and a work breakdown of the effort are also presented.

14. **SUBJECT TERMS:** model checking, systems upgrade, model-based verification, tool analysis, system model, specified claims, logic formulae, software systems, modeling processes

15. **NUMBER OF PAGES**
54

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102